加餐2 | 用户Kaito：我是怎么读Redis源码的？

2021-09-18 Kaito

《Redis源码剖析与实战》 课程介绍 >

讲述：正霖

时长 15:04 大小 13.80M

你好，我是 Kaito，也是两季 Redis 课程的课代表。今天，我想来和你分享一下我读源码的经验，希望能助力你更好地学习 Redis 源码。

首先，一提到读源码，很多人都会比较畏惧，认为读源码是高手才会做的事情。可能遇到问题时，他们更倾向于去找别人分享的答案。但往往很多时候，自己查到的资料并不能帮助解决所有问题，尤其是比较细节的问题。

那么从我的实践经验来看，遇到这种情况，通常就需要去源码中寻找答案了，因为在源码面前，这些细节会变得一览无余。而且我认为，掌握读源码的能力，是从只懂得如何使用 Redis，到精通 Redis 实现原理的成长之路上，必须跨越的门槛。可是，**面对庞大复杂的项目，我们怎样读源码才能更高效呢？**

所以下面，我就来和你聊一聊我在读源码时的经验和心得。

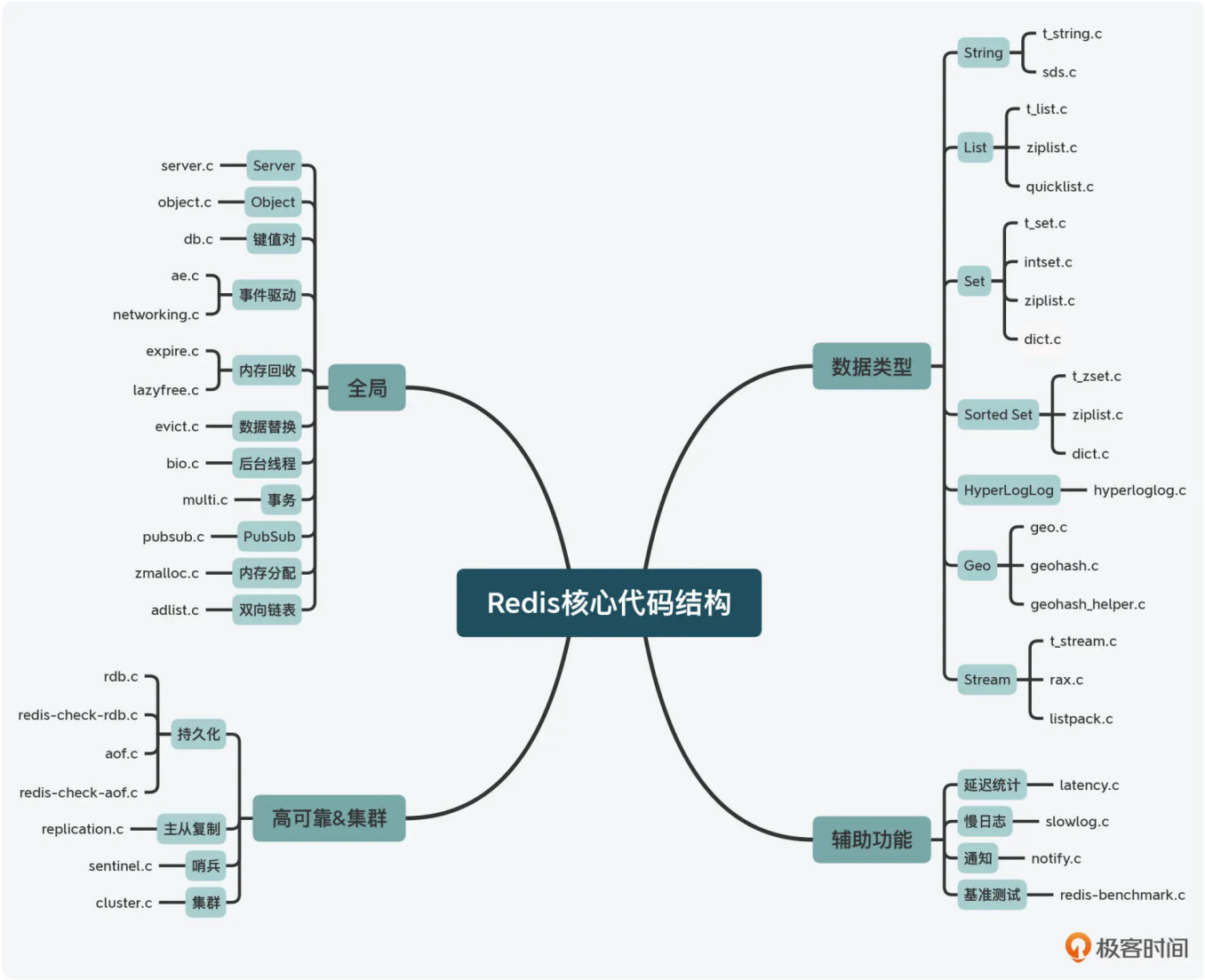
找到地图

很多开源项目的源码，代码量一般都比较庞大，如果在读代码之前，我们没有制定合理的方法，就一头扎进去读代码，势必会把自己搞晕。

所以，我在拿到一个项目的代码之后，并不会马上着手去读，而是会先对整个项目结构进行梳理，划分出项目具体包含的模块。这样，我就对整个项目有了一个宏观的了解。

因为读代码就好比去一个陌生城市旅行，这个旅途过程充满着未知。如果在出发之前，我们手里能有一张地图，那我们对自己的行程就可以有一个非常清晰的规划，我们就知道，如果想要到达目的地，需要从哪里出发、经过哪些地方、通过什么方式才能到达，**有了地图就有了行进方向**，否则很容易迷失。

因此，提前花一些时间梳理整个项目的结构和目录，对于后面更好地阅读代码是非常有必要的。就拿 Redis 来举例，在读 Redis 源码之前，我们可以先梳理出整个项目的功能模块，以及每个模块对应的代码文件，以下给出的就是 src 下的代码结构：



这样，有了这张地图之后，我们再去看代码的时候，就可以有重点地阅读了。

前置知识准备

在梳理完整个项目结构之后，我们就可以正式进入阅读环节当中了。

不过，在阅读代码之前，我们其实还需要预先掌握一些前置知识。因为一个完整的项目，必然综合了各个领域的技术知识点，比如数据结构、操作系统、网络协议、编程语言等，如果我们提前做好一些功课，在读源码的过程中就会轻松很多。以下是根据我在阅读 Redis 书籍和实战过程中，提取的读源码必备前置知识点，给你参考一下。

常用数据结构：数组、链表、哈希表、跳表。

网络协议：TCP 协议。

网络 IO 模型：IO 多路复用、非阻塞 IO、Reactor 网络模型。

操作系统：写时复制（Copy On Write）、常见系统调用、磁盘 IO 机制。

C 语言基础：循环、分支、结构体、指针。

当然，在阅读源码的过程中，我们也可以根据实际问题再去查阅相关资料，但不管怎样，提前熟悉这些方面的知识，在真正读代码时就会省下不少时间。

从基础模块开始读

好，有了地图并掌握了前置知识之后，接下来我们就要进入主题了：**读代码**。但具体要从哪个地方开始读起呢？我认为要先从最基础的模块开始读起。

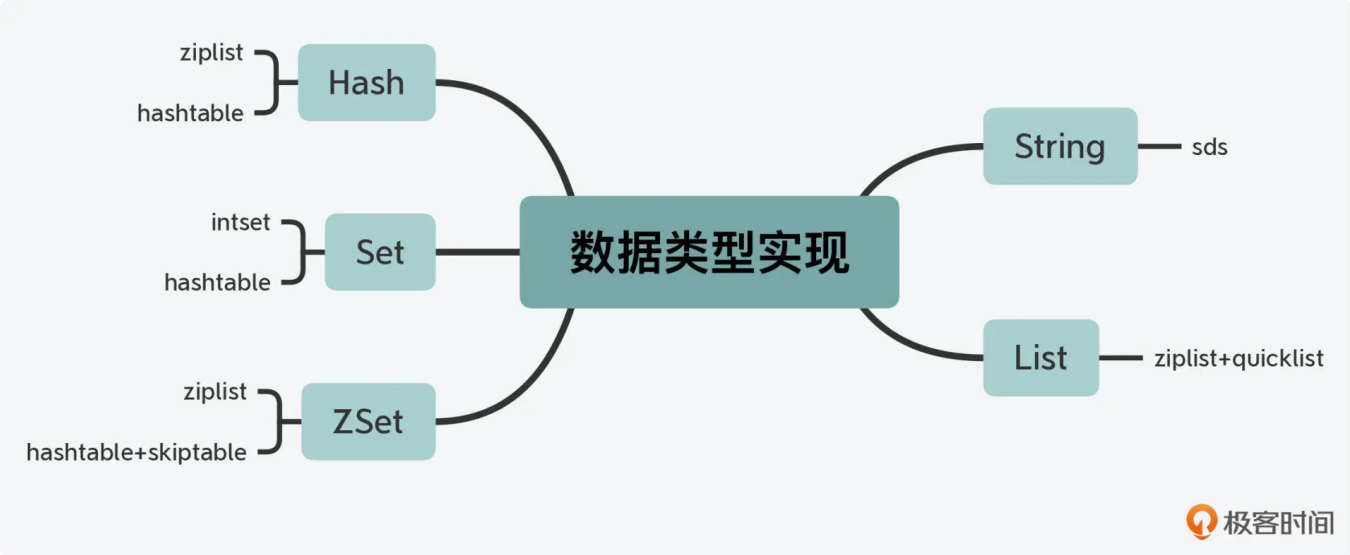
我在前面也分析了，一个完整的项目会划分很多的功能模块，但这些模块并不是孤立的，而很可能是有依赖关系的。比如说，Redis 中的 networking.c 文件，表示处理网络 IO 的具体实现。而如果我们能在理解事件驱动模块 ae.c 的基础上，再去阅读网络 IO 模块，效率就会更高。

那么在 Redis 源码中，哪些是最基础的模块呢？想一下，我们在使用 Redis 时，接触最频繁的是哪些功能？

答案是**各种数据类型**。

一切操作的基础，其实都是基于这些最常用的数据类型来做的，比如 String、List、Hash、Set、Sorted Set 等。所以，我们就可以从这些基础模块开始读起，也就是从 t_string.c、t_list.c、t_hash.c、t_set.c、t_zset.c 代码入手。

如果你对 Redis 的数据类型有所了解，就会看到这些数据类型在实现时，底层都对应了不同的数据结构。比如，String 的底层是 SDS，List 的底层是 ziplist + quicklist，Hash 底层可能是 ziplist，也可能是哈希表，等等。



而由此一来，我们会发现，这些数据结构又是更为底层的模块，所以我们在阅读数据类型模块时，就需要重点聚焦在这些模块上，也就是 sds.c、ziplist.c、quicklist.c、dict.c、intset.c 文件，而且这些文件都是比较独立的，阅读起来就可以更加集中。

这样，当我们真正掌握了这些底层数据结构的实现后，就能更好地理解基于它们实现的各种数据类型了。**这些基础模块就相当于一座大厦的地基，地基打好了，才能做到高楼耸立。**

找到核心主线

接着，掌握了数据结构模块之后，这时我们的重点就需要放在核心主线上来了。

在这个阶段，我们需要找到一个明确的目标，以这个目标为主线去读代码。因为读源码其实是一个很常见的需求，就是为了了解这个项目最核心功能的实现细节，我们只有以此为目标，找到这条主线去读代码，才能达到最终目的。

那么在读 Redis 源码时，什么才是它的核心主线呢？这里我分享一个非常好用的技巧，就是根据“**Redis 究竟是怎么处理客户端发来的命令的？**”来梳理。

举个例子，当我们在执行 SET testkey testval EX 60 这样一条命令时，就需要搞清楚 Redis 是怎么执行这条命令的。也就是要明确，Redis 从收到客户端请求，到把数据存到 Redis 中、设置过期时间，最后把响应结果返回给客户端，整个过程的每一个环节，到底是如何处理的。

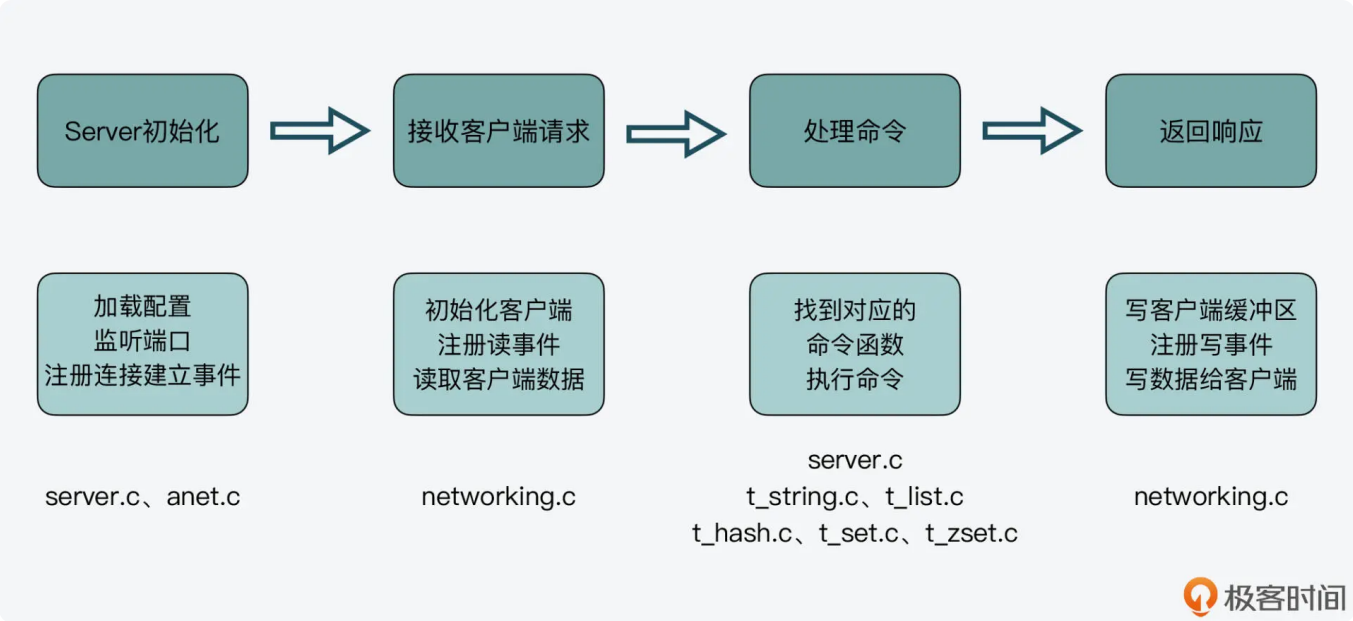
有了这条主线，我们就有了非常明确的目标，而且沿着这条主线去读代码，我们还可以很清晰地把多个模块串联起来。比如从前面的例子中，我们会看到一条命令的执行，主要包含了这样几个阶段。

Redis Server 初始化：加载配置、监听端口、注册连接建立事件、启动事件循环（server.c、anet.c）。

接收、解析客户端请求：初始化 Client、注册读事件、读客户端 Socket（networking.c）。

处理具体的命令：找到对应的命令函数、执行命令（server.c、t_string.c、t_list.c、t_hash.c、t_set.c、t_zset.c）。

返回响应给客户端：写客户端缓冲区、注册写事件、写客户端 Socket（networking.c）。



沿着这条主线去读代码，我们就可以掌握一条命令的执行全过程。而且，由于这条主线的代码逻辑，已经覆盖了所有命令的执行流程，我们下次再去读其他命令时，比如 SADD，就只需要关注处理命令部分的逻辑即可，其他逻辑有 80% 都是相同的。

先整体后细节

当然，在阅读主线代码的过程中，肯定也会遇到过于复杂的函数，第一次在读这种函数时，很容易就会陷进去，导致整个主线代码的阅读，无法继续推进下去。遇到这种情况其实是很正常的，可这时我们应该怎么办呢？

这里我的做法是，前期读到这种逻辑时，不要马上陷入到细节中去，而是要**先抓整体**。具体来说，对于复杂的函数逻辑，我们刚开始并不需要知道它的每一个细节是如何实现的，而是只需知道这个函数大致做了几件事情即可。

举个例子，在执行 HSET 命令时，有一段代码很复杂，其中包括了很多分支判断，一次很难读懂：

复制代码

```
int hashTypeSet(robj *o, sds field, sds value, int flags) {
    int update = 0;

    if (o->encoding == OBJ_ENCODING_ZIPLIST) {
        unsigned char *zl, *fptr, *vptr;

        zl = o->ptr;
        fptr = ziplistIndex(zl, ZIPLIST_HEAD);
        if (fptr != NULL) {
            fptr = ziplistFind(fptr, (unsigned char*)field, sdslen(field), 1);
            if (fptr != NULL) {
                /* Grab pointer to the value (fptr points to the field) */
                vptr = ziplistNext(zl, fptr);
                serverAssert(vptr != NULL);
                update = 1;

                /* Delete value */
                zl = ziplistDelete(zl, &vptr);

                /* Insert new value */
                zl = ziplistInsert(zl, vptr, (unsigned char*)value,
                        sdslen(value));
            }
        }

        if (!update) {
            /* Push new field/value pair onto the tail of the ziplist */
            zl = ziplistPush(zl, (unsigned char*)field, sdslen(field),
                    ZIPLIST_TAIL);
            zl = ziplistPush(zl, (unsigned char*)value, sdslen(value),
                    ZIPLIST_TAIL);
        }
        o->ptr = zl;

        /* Check if the ziplist needs to be converted to a hash table */
        if (hashTypeLength(o) > server.hash_max_ziplist_entries)
            hashTypeConvert(o, OBJ_ENCODING_HT);
    } else if (o->encoding == OBJ_ENCODING_HT) {
        dictEntry *de = dictFind(o->ptr,field);
        if (de) {
```

```
            sdsfree(dictGetVal(de));
            if (flags & HASH_SET_TAKE_VALUE) {
                dictGetVal(de) = value;
                value = NULL;
            } else {
                dictGetVal(de) = sdsdup(value);
            }
            update = 1;
        } else {
            sds f,v;
            if (flags & HASH_SET_TAKE_FIELD) {
                f = field;
                field = NULL;
            } else {
                f = sdsdup(field);
            }
            if (flags & HASH_SET_TAKE_VALUE) {
                v = value;
                value = NULL;
            } else {
                v = sdsdup(value);
            }
            dictAdd(o->ptr,f,v);
        }
    } else {
        serverPanic("Unknown hash encoding");
    }

    /* Free SDS strings we did not referenced elsewhere if the flags
     * want this function to be responsible. */
    if (flags & HASH_SET_TAKE_FIELD && field) sdsfree(field);
    if (flags & HASH_SET_TAKE_VALUE && value) sdsfree(value);
    return update;
}
```

那么，我在读这段代码时，就可以先简化逻辑，把握整体思路：

复制代码

```
// 执行 HSET 命令
int hashTypeSet(robj *o, sds field, sds value, int flags) {
    ...
    // 如果当前是 ziplist 编码存储
    if (o->encoding == OBJ_ENCODING_ZIPLIST) {
        ...
    // 如果是 hashtable 编码存储
    } else if (o->encoding == OBJ_ENCODING_HT) {
        ...
    } else {
        serverPanic("Unknown hash encoding");
```

```
    }
    ...
}
```

之后，再了解每个分支大致做了哪些事情：

复制代码

```
// 如果当前是 ziplist 编码存储
if (o->encoding == OBJ_ENCODING_ZIPLIST) {
    ...
    // field 在 ziplist 中，从 ziplist 中删除，在插入新值
    if (fptr != NULL) {
        fptr = ziplistFind(fptr, (unsigned char*)field, sdslen(field), 1);
        zl = ziplistDelete(zl, &vptr);
        zl = ziplistInsert(zl, vptr, (unsigned char*)value, sdslen(value));
        ...
    }

    // field 不在 ziplist 中，则插入到 ziplist
    if (!update) {
        zl = ziplistPush(zl, (unsigned char*)field, sdslen(field),
                         ZIPLIST_TAIL);
        ...
    }

    // ziplist 元素超过阈值，转为 hashtable
    if (hashTypeLength(o) > server.hash_max_ziplist_entries) {
        hashTypeConvert(o, OBJ_ENCODING_HT);
    }
}
```

这样做的**好处**，一是不会被复杂的细节逻辑搞晕，打击自己的自信心；二是可以有效避免阅读的连贯性被打断，从而能持续推进我们把整个主线逻辑读完。

所以，这里的重点就是：先把复杂代码的主逻辑搞清楚，知道涉及的每个方法完成了什么事，心里要先搭建一个简单的框架，等有了框架之后，我们再去给框架填充细节。这样通过**先整体后细节**的方式，我们就可以不再畏惧代码中的复杂逻辑。

先主线后支线

不过，在阅读主线代码的过程中，我们肯定还会遇到各种支线逻辑，比如数据过期、替换淘汰、持久化、主从复制等。

其实，在阅读主线逻辑的时候，我们并不需要去重点关注这些支线，而当整个主线逻辑清晰起来之后，我们再去读这些支线模块，就会容易很多了。这时，我们就可以从这些支线中，选取下一个目标，带着这个目标去阅读，比如说：

过期策略是怎么实现的？（expire.c、lazyfree.c）

淘汰策略是如何实现的？（evict.c）

持久化 RDB、AOF 是怎么做的？（rdb.c、aof.c）

主从复制是怎么做的？（replication.c）

哨兵如何完成故障自动切换？（sentinel.c）

分片逻辑如何实现？（cluster.c）

....

有了新的支线目标后，我们依旧可以采用前面提到的先整体后细节的思路阅读相关模块，这样下来，整个项目的每个模块，就可以被逐一击破了。

查漏补缺

最后，我们还需要查漏补缺。

按照前面提到的方法，基本就可以把整个项目的主要模块读得七七八八了，这时我们基本已经对整个项目有了整体的把控。

不过，当我们在工作中遇到问题时，很有可能会发现，在当时读代码的过程中，有很多并不在意的细节被忽略了。所以这时，我们就可以**再带着具体问题出发，聚焦这个问题相关的模块，再一次去读源码**。这样一来，我们就可以填补当时阅读源码的空白区。

举个例子，当我们在阅读 String 底层数据结构 SDS（简单动态字符串）的实现时，我们会看到当 SDS 需要追加新内容时会进行扩容，而我们之前阅读这部分代码时，**很有可能只是了解到有这样的逻辑存在，但并没有在意扩容的相关细节（一次扩容多大）**。

所以，当我们在工作中遇到这个细节问题后，就可以把目光聚焦在 SDS 的扩容逻辑上（sds.c 的 sdsMakeRoomFor 函数），而此时我们会发现，当需要申请的新内存小于 1MB 时，Redis 就会翻倍申请内存，否则按 1MB 申请新内存。

采用这个方法进行查漏补缺，我们就可以对整个项目了解得更深入、更全面，真正把项目吃透。

总结

好了，以上就是我在阅读 Redis 源码时，总结出来的经验体会，这里我也把这七个步骤梳理总结下。

1. **找到地图**：拿到项目代码后，提前梳理整个项目结构，知晓整个项目的模块划分，以及对应的代码文件。
2. **前置知识准备**：提前掌握项目中用到的前置知识，比如数据结构、操作系统原理、网络协议、网络 IO 模型、编程语言语法等等。
3. **从基础模块开始读**：从最底层的基础模块开始入手，先掌握了这些模块，之后基于它们构建的模块读起来会更加高效。
4. **找到核心主线**：找到整个项目中最核心的主线逻辑，以此为目标，了解各模块为了完成这个功能，是如何协作和组织的。
5. **先整体后细节**：对于复杂函数，不要上来就陷入细节，前期阅读只需了解这个函数大致做了什么事情，建立框架，等搭建起框架之后，再去填充细节。
6. **先主线后支线**：整个主线逻辑清晰之后，再去延伸阅读支线逻辑，因为支线逻辑肯定是服务主线逻辑的，读完主线后再去读这些支线，也会变得更简单。
7. **查漏补缺**：在工作中遇到具体问题，带着这些实际的问题出发再次去读源码，进行查漏补缺，填补之前读源码时没有注意到的地方。

当然，这个阅读源码的方法也并不局限于 Redis，如果你在开发过程中，也有读源码的需要，希望这个方法能帮助你更好地吃透它。

最后，如果你也有自己的阅读源码的实践经验和方法，欢迎在留言区分享出来，我们一起交流，共同进步！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 2 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 加餐1 | Redis性能测试工具的使用

下一篇 18 | 如何生成和解读RDB文件？

更多学习推荐

精选留言 (5)

写留言

Kaito
2021-09-18

很荣幸，能在第二季继续给大家带来加餐文章，分享我学习Redis的经验。

这篇文章和大家分享我阅读源码的经验和心得，当然，文章里提到的方法是「通用」的，不仅限于读Redis代码，读任何项目的源码都可以按这个思路来，希望我的分享能够帮助到大家！

展开

2 6

Milittle
2021-09-18

这个专栏可以说是有两个老师。

展开

1

Ethan New

2021-09-19

感谢Kaito大佬的分享，向大佬学习

展开

Bruis.

2021-09-18

Kaito大哥，如何对Redis源码进行编译，并能在开发工具上进行debug调试呢？光看源码不debug运行感觉很多细节是没法读懂的啊。

展开

曾轼麟

2021-09-18

感谢Kaito同学的分享，这段时间受益良多

展开